DSL course: individual work #1

Tasks

- [x] create data structure for regular expression syntax tree;
- [x] function to parse expression into AST;
- [x] create data structure for non-deterministic automata;
- [x] create function that transforms regular expression AST into automata;
- [x] create an application that receives the representation of regular expression and some expression and checks whether the statement satisfies the regexp condition.
- [ ] create data structure for deterministic automata;
- [ ] create algorithm for determinization(with deletion of dead-end and unreachable states);
- [ ] create algorithm for minimization(reduction);
- [ ] create algorithm for defragmentation(do index numbers of states consistent).

How to install and run

Clone/download the repository and execute `python lab1_app.py`. There is no external dependencies, only standard types and type annotations.

It works on `python3.6`, so please use the same or newer version.

Test set

There are some test sequences inside `test_cases.txt`. Just run `lab1_app.py` and input the filename.

There is `tests.py`, run it, it has other checks and examples. To understand what is going on there, open the script is required.

About the implementation

Syntax of the regular expressions

- `a` — just any symbol, UTF-8 symbols can be used;

- `|` — logical `or` statement (priority **1**);
- `ab` — is `a+b` there `+` is concatenation (priority **2**);
- `a*` — `*` is Clini closure (priority **3**), it means that expression to which it is performed cannot occur or occur any number of times;
- parenthesis `(` and `)`, as usual, performs prioritization of some subexpression.

So, the following expressions and like them are allowed:

- `(a)(b)`
- `(ab)*`
- `a|(b*c*)*`

Escaped symbols

- `\*`
- `\(`
- `\)`
- `\|`

If the symbols are escaped, they can be used as other symbols, not a special ones.
As example: regexp `\(ab\)\*` can be satisfied by only expression `(ab)*`.

How to use the `app.py`

Create file with your regular expressions and expressions that should be checked in the following format:

- every line should contain 2 or 3 substrings with colon (`:`) delimiter;
- the first substring is a regular expression. Be careful, for the regular expressions all symbols matter (including spacing);
- second and third substrings are expressions that should be checked;
- first of them are expressions that must match the regexp;
- second of them are expressions that must not match the regexp;
- all expressions for checking separated by a semicolon(`;`).
- you can keep the first list of expressions empty, but colons should be placed.

So, the typical line of the testing file can be:

- `(ab)*:;ab;abab:aa;bb` — regexp + true + false expressions
- `hii*:hi;hiiii` — regexp + only true expressions
- `ab*::b;bb;ba` — regexp + empty + only false expressions

It means, that you can't use colon and semicolon in your expressions, but it is the limitation of the current verifying application, not of the regexp or state machine implementation.

About state machines

`automata` package implements non-deterministic finite state machine.
It means that amount of states are finite and from the same state
can exist more than one transition with the same trigger symbol.

Transitions are stored as dictionary of dictionaries of sets (or in Python notation `Dict[int, Dict[chr, Set[int]]]`), each machine can have only one start state (always `0`), and any number of final states.

Forks with transitions of the same trigger symbol are allowed. In the machine, this case is handled by multiple current states of the machine (to be exact, set of states). If fork occurs while executing and transitions of the same symbol are performed, current state will be split up into multiple states, every of which will continues its route on its branch.

About regular expression parsing

After scanning (it looks really simple), parsing is performed. List of tokens is recursively separated by one of the above operators with respect to their priority.

Note:
All the actions occur on the "first layer" of the expression, and here is what I mean.

Let's say that we have regular expression `(ab)|(c|d)`. The "first layer" of the expression is `group1|group2`, where `group1` is `(ab)`, `group2` is `(c|d)` and `|` is binary operator. Every iteration of the algorithm must work only with the parts of the first layer (operations inside groups are not visible by algorithm).